Assignment 6: Latex G5

Group 5: Alexandros Tsiridis, Stamatis Maritsas, Andray Afanasyev

Contents

List of Figures

List of Tables

1 XML - XPointer

Stamatis Maritsas

1.1 Introduction

XPointer is a system that allows the creation of links that point to components of XML. It is divided among four specifications:

1. XPointer framework
2. XPointer element() scheme

3. XPointer xmlns() scheme
4. XPointer xpointer() scheme

The above points are going to be explained further to the subsections below. It is worth mentioning that XPointer Framework is a W3C recommendation since March 2003.

1.2 XPointer framework

The XPointer framework's specification defines the XML Pointer Language (XPointer), which is an extensible system for XML addressing and includes the XPointer schemes that mentioned in the introduction.

This framework can be used as a specification for creating XML fragment identifiers for resources whose media type is one of text/xml, application/xml, text/xml-external-parsed-entity, or application/xml-external-parsed-entity. In addition, it is also encouraged to use this framework to define your own fragment identifier language. [4]

1.3 XPointer element() scheme

This kind of XPointer scheme is tend to be used to allow basic addressing of XML elements. Furthermore, the formal grammar for this scheme is given using simple EBNF (Extented Backus-Naur) notation as described in the XML Recommendation.[3]

1.4 XPointer xmlns() scheme

The purpose of this scheme is to interpret the namespace prefixes in pointers correctly. For example, a developer can create his own namespaces and place them in an XML document and then use the xmlns() scheme to point to those namespaces. Just like the XPointer element() scheme, the formal grammar for the xmlns() scheme is given using simple EBNF notation, as described in the XML Recommendation.[2]

1.5 XPointer xpointer() scheme

The role of this scheme in the whole XML XPointer philosophy is to address efficiently fragments of XML documents. In order to achieve this goal it uses the XPath and adds the ability to address strings, points and ranges. It supports addressing not only into the internal structures of XML documents but also to external parsed entities. Moreover, it allows the examination of a document's hierarchical structure and choice of portions based on various properties (i.e. element types, etc.). Another thing that I have to mention is the fact that, the xpointer() scheme's extensions to XPath provide the ability to identify locations that are not single, whole elements and combine string matching with the other location methods supplied.[1]

1.6 XML XPointer in practice

There are two ways of linking a part of an XML document:

- We can simply add a number sign (#) and an XPointer expression after the URL in the xlink:href attribute, as follows: *xlink:href="http://dog.com/dogbreeds.xml#xpointer(id('Rottweiler'))"*. The expression refers to the element in the target document, with the id value of "Rottweiler".
- XPointer also allows a shorthand method for linking to an element with an id. You can use the value of the id directly, like this: *xlink:href="http://dog.com/dogbreeds.xml#Rottweiler"*.

XML File

```
<?xml version="1.0" encoding="UTF-8"?>

<dogbreeds>

<dog breed="Rottweiler" id="Rottweiler">
  <picture url="http://dog.com/rottweiler.gif" />
  <history>The Rottweiler's ancestors were
  probably Roman drover dogs.....</history>
  <temperament>Confident, bold, alert and imposing,
  the Rottweiler is a popular choice for its
  ability to protect....</temperament>
</dog>

<dog breed="FCRetriever" id="FCRetriever">
  <picture url="http://dog.com/fcretriever.gif" />
  <history>One of the earliest uses of retrieving
  dogs was to  help fishermen retrieve fish from
  the water....</history>
  <temperament>The flat-coated retriever is a
  sweet, exuberant,  lively dog that loves to play
  and retrieve....</temperament>
</dog>
</dogbreeds>
```

XML file with Xpointers

```
<?xml version="1.0" encoding="UTF-8"?>
<mydogs xmlns:xlink="http://www.w3.org/1999/xlink">

<mydog>
  <description>
  Anton is my favorite dog. He has won a lot of.....
  </description>
  <fact xlink:type="simple"
  xlink:href="http://dog.com/
  dogbreeds.xml#xpointer(id('Rottweiler'))">
  Fact about Rottweiler
  </fact>
</mydog>

<mydog>
  <description>
  Pluto is the sweetest dog on earth......
  </description>
  <fact xlink:type="simple"
  xlink:href="http://dog.com/dogbreeds.xml#FCRetriever">
  Fact about flat-coated Retriever
  </fact>
</mydog>

</mydogs>
```

Figure 1: Example code of the two ways (modified version) [5]

1.7 XPointer VS XPath

XPointer	XPath
same expression	same expression
specifies the connections of URIs	says nothing about URIs
new features introduced	no available features

Table 1: Comparison of XPointer and XPath

2 Overhauling Amd for the 00s: A Case Study of GNU Autotools

2.1 Introduction

As group 5 we were assigned to look at the above article and discuss it during the lecture. In the following section, I am going to discuss about the article and what information I believe is useful to grasp from the paper, and what I learned from it. First of all, I will break up the article in three parts:

1. How the build was done before the **autotools** and what autotools have to offer.
2. Measuring complexity.
3. What you can gain by the use of **autotools**.

2.2 How the build was done before the autotools and what autotools have to offer

Without the use of **autotools**, the building and configuration of the package is being done manually by the user and the programmer of it. At the first step, users have to manually configure a package

before the compilation by changing the header file. In order to achieve this goal, the user need to have intimate knowledge of the system that he was trying to install the package on.

Moreover, in order for the programmers to achieve desired compatibility for the packages they are making use of extended CPP macros. These macros are being used to recognize features that the operating system need to have due to them being dependencies of the package. Such macros are complicated to write and usually end up in an extended nested form. Due to the complexity of the macros and their nested forms, maintenance of the package by the programmers becomes a nightmare.

Another way that is being used is the Imake utility which is designed specifically for building X11 applications. This tool defines static configurations for various systems, which cannot be account for local changes made by the user. One more utility which is being used is Metaconfig. This utility executes simple tests to find out the features of the system, but needs a lot of interaction with the user to confirm that the detection is correct.

GNU autotools were created in order to address the problems that were mentioned above. This is achieved by providing in build tests to dynamically detect various features of the system that are needed for the package. It is a suite of three tools namely *Autoconf, Automake and Libtool.*

2.3 Measuring complexity

In this subsection I am going to talk about three ways of measuring complexity that were mentioned in the article. I find these information quite useful even if someone is not interested in **autotools** or how to build a package without the use of them. In the paper, the three ways of measurement are:

- Measuring complexity according to the amount of the code lines inside the package.
- Measuring complexity according to the amount of total CPP macros across the whole package.
- Measuring complexity according to the average of total CPP macros per 1000 lines of code.

The first measurement will obviously make the biggest package the most complex. In my opinion, this is not an accurate way of measuring complexity as a bigger package might actually be less complex than a smaller one.

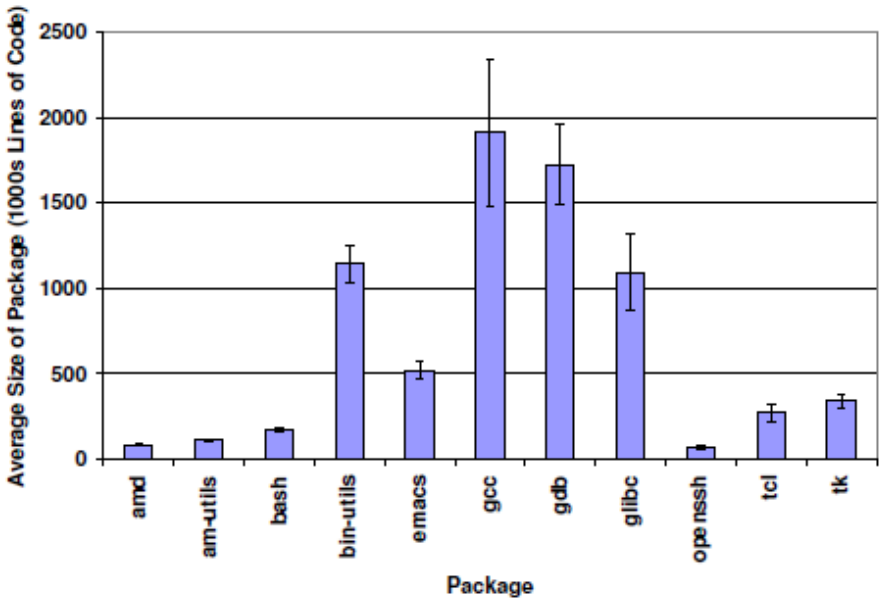

Figure 2: Average size of packages in thousands of lines of code [6]

The second measurement its a bit more accurate than the way discussed above. It also has a fault as bigger packages usually tend to have more CPP conditionals than the smaller ones making them candidates for being the most complex ones in this measurement.

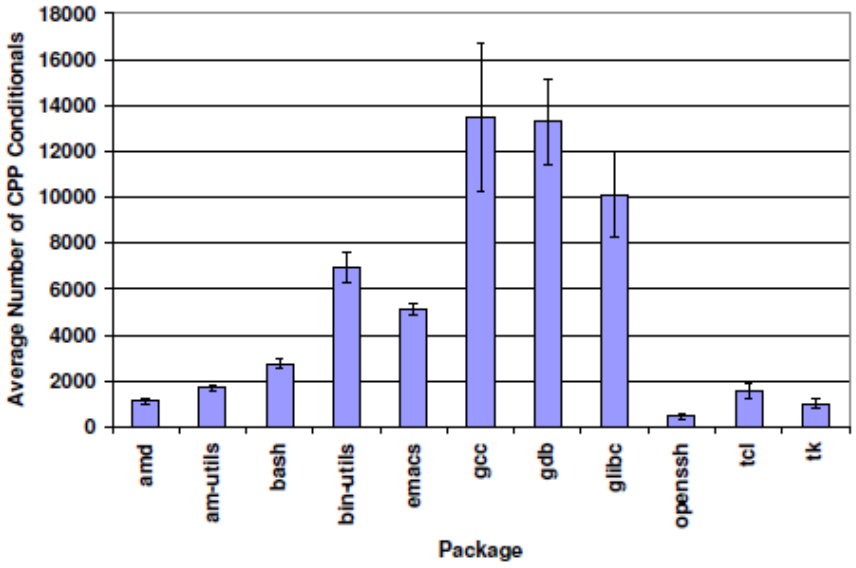

Figure 3: Average number of CPP conditionals per package [6]

The third measurement in my opinion, is better than the above two as it actually counts how often CPP commands are introduced inside the code.

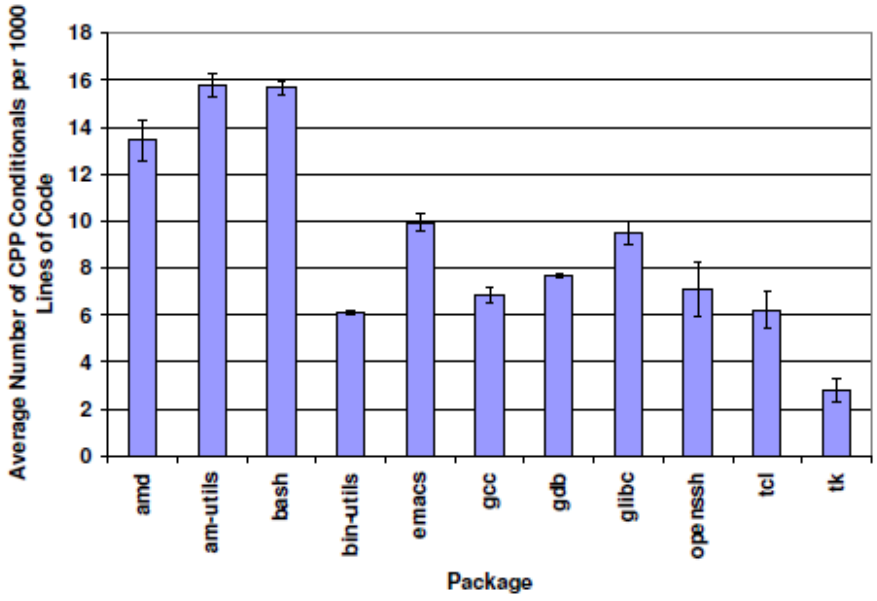

Figure 4: Average number of CPP conditionals per 1000 lines of code [6]

To sum up, let's take the packages example seeing in the figures of gcc and amd. In the first figure, gcc climbs to the top of the complexity scale as the biggest package and amd is second from last due to its small size. In the second measurement, gcc is also the most complex one by a small difference from gdb and amd is third from last. And at the last measurement, where the size of the package does not actually matter, gcc is fourth in complexity from last and amd climbs to the third most complex package.

2.4 What you can gain by the use of autotools

By the use of **autotools** there are various benefits as well as a few disadvantages that are introduced. First of all, they reduce the complexity of code by automating the feature discovery procedure. They minimize the use of CPP conditionals by providing those in build tests. On the other hand, they do not reduce to zero the usage of those macros as they do not cover all the possible tests that the package might need to do. BY the reduction of complexity, as shown in the article, programmers become more rapid developers as they introduce more features in less time than they did without the use of **autotools**. In addition, the programs are easier to maintain by the developers and easier to be installed by the users as the process of building and compiling the package becomes automated.

On the other side of the coin, **autotools** are not perfect and they have their disadvantages. Firstly, the developers need to be well-trained and used to these tools. This procedure, of learning the tools, takes a significant amount of time as stated at the article, it took them 54 months to learn how to use **autotools** properly. Secondly, **autotools** are making the compilation significantly slower as stated in the article, it took three times more time to compile their package with **autotools** than the version that did not use **autotools**. *The result of the test performed on the article can be seen at the following table:*

Action and Package	Time
Build Amd-upl102	35.4
Configure Am-utils-6.0a1 (no cache)	102.6
Configure Am-utils-6.0a1 (with cache)	24.2
Compile Am-utils-6.0a1	73.1

Table 2: Time (seconds) to Configure and Build Amd Packages [6]

In my opinion, **autotools** are useful to be learned by companies that produce a lot of software as the process of learning the **autotools** will provide them with less complex programs in the future. In addition, companies that provide long term packages will also be benefit by using those tools.

References

[1] Steve DeRose, Eve Maler, and Ron Daniel. Xpointer xpointer () scheme. *W3C Working Draft*, 19, 2002.

[2] Steven J DeRose, Ron Daniel Jr, Eve Maler, and Jonathan Marsh. Xpointer xmlns () scheme. *W3C Recommendation*, 25, 2003.

[3] Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. Xpointer element () scheme. *World Wide Web Consortium, Recommendation REC-xptr-element-20030325*, 2003.

[4] Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. Xpointer framework. *W3c recommendation*, 25, 2003.

[5] W3Scools.com. Xml, xlink and xpointer. `http://www.w3schools.com/xml/xml_xlink.asp`. Online; accessed 26-September-2015.

[6] Erez Zadok. Overhauling amd for the'00s: A case study of gnu autotools. In *USENIX Annual Technical Conference, FREENIX Track*, pages 287–297, 2002.